Coalition structure generation problems: optimization and parallelization of the IDP algorithm

Francisco Cruz[1,2], Jesús Cerquides[2], Antonio Espinosa[1], Juan Carlos Moure[1], Sarvapali D. Ramchurn[3], and Juan Antonio Rodriguez-Aguilar[2]

[1] Computer Architecture Department. Universitat Autònoma de Barcelona. Spain
[2] Institut d'Investigació en Intel·ligència Artificial -CSIC- Spain
[3] School of Electronics and Computer Science, University of Southampton, Southampton

Abstract. The Coalition Structure Generation (CSG) problem is well-known in the area of Multi-Agent Systems. Its goal is to establish coalitions between agents while maximizing the global welfare. Between the existing different algorithms designed to solve the CSG problem, DP and IDP are the ones with smaller temporal complexity. After analyzing the operation of the DP and IDP algorithms, we identify which are the most frequent operations and propose an optimized method. Then, we analyze the memory access pattern and find that its irregular behavior represents a potential performance bottleneck. In addition, we study and implement a method for dividing the work in different threads. We show that selecting the best algorithmic options can improve performance by 10x or more. Furthermore, the execution in a dual-socket, six-core processor computer may increase performance by an additional 5x-6x. With this, we are able to solve a CSG problem of 27 agents using our multi-core computer in 1.2 hours.

1 Introduction

Coalition formation is one of the central types of interaction in multi-agent systems. It involves the creation of disjoint groups of autonomous agents that collaborate in order to satisfy their individual or collective goals. One of the major research challenges in the field of multi-agent systems is the search for an effective set of coalition that maximises the social welfare [1].

Coalition formation can be applied to many actual-world problems such as distributed vehicle route planning [2], task allocation [1], airport slots allocation [3]. More recently, it has been considered in the realm of social networks [4].

According to [2], the coalition formation process is divided in three activities [5]: (i) coalition structure generation; (ii) solving the optimization problem of each coalition; and (iii) dividing the value of the generated solution among agents. In this paper we focus on one of these three activities, namely coalition structure generation (CSG). Notice that finding the optimal coalition structure

is $\mathcal{NP}$-complete [2]. The search space handled by CSG is very large since the number of possible coalitions grows exponentially with the number of agents (n).

To tackle the CSG problem, we find several algorithms in the literature that take different perspectives. In particular, they are typically grouped in three categories: (i) lower-complexity ($O(3^n)$) complete algorithms based on dynamic programming (e.g. DP [6], IDP [7]), which offer guaranteed run-times over arbitrary coalition value distributions; (ii) higher-complexity ($O(n^n)$) complete algorithms (e.g. IP[8] IP-IDP [9], D-IP [10] with anytime properties whose convergence time to solution largely depends on the coalition value distribution; and (iii) heuristic approximate algorithms (e.g. [1]), which aim at computing solutions faster than complete algorithms without offering quality guarantees. Unfortunately, as widely noticed in the literature, in practice the computational costs for complete algorithms are highly demanding even for a moderate number of agents (e.g. IDP requires around 2.5 days for 27 agents [8]).

Against this background, in this paper we propose to optimize the algorithms based on dynamic programing. The implementation can be used as a building block for heuristic algorithms as a means to explore complete subspaces in an effective way.

As proposed in D-IP [10], where a distributed anytime algorithm is presented, in this paper we present an algorithm able to exploit the power of distribution but using a different paradigm. Our proposal is building a IDP based algorithm able to run in a shared memory scenario, which is common in nowadays computers [11]. Using a shared memory paradigm simplifies the communication between computation nodes, since there is no need to send messages between them, but it requires a data dependence study, because of possible synchronization.

As far as we are concerned, no reference implementation neither of DP nor IDP algorithms has been published. When studying and evaluating different implementation alternatives, we have found, though, non-negligible issues on the algorithmic details that have a considerable impact on the overall performance. The contributions of this work can be summarized as:

- We analyze and evaluate alternative fast methods for the most critical operation, establishing that a bad choice can degrade performance by $10x$ or more.
- We parallelize the generation of splittings, the most time-consuming operation, and execute the problem on a shared-memory, multi-core, multi-thread and multi-processor system.
- We identify the main performance bottleneck: both the sequential and parallel execution are limited by the lack of temporal and spatial locality of the memory access pattern, and by the weak support for irregular and scattered accesses provided by current memory hierarchies.
- We find out that the performance advantage of IDP versus DP is only realized for large problems, when reducing memory bandwidth requirements pay off.
- We make our code publicly available at the following URL: https://github.com/CoalitionStructureGeneration/DPIDP.

The paper is organized as follows. Section 2 introduces the CSG problem and describes the state of the art on dynamic programming techniques. Section 3 analyzes implementation issues such as data representation, most frequent operations and bottlenecks in a single core environment and proposes solutions to reduce execution time. Section 4 studies how to parallelize the IDP algorithm and Section 5 evaluates the performance of single and multi-threaded implementations. Finally, Section 6 concludes and outlines future work.

2 The coalition structure generation problem

In this section we describe what a Coalition Structure Generation (CSG) problem is and how dynamic programming algorithms have addressed it to find an optimal solution.

The CSG involves partitioning the set of all agents so as to maximise the sum of the values of the chosen coalitions.

Considering a group of agents, the value associated to every possible coalition is known or can be calculated. This set of values can be stored in a table. Table 1 presents an example of the input data for a CSG problem among 4 agents. The goal of the CSG problem is to find the combination of disjoint coalitions that maximize the global value by aggregation. From Table 1 it is easy to identify some preferred coalitions, for example it is obvious that the coalition formed by $\{a_2,a_3\}$ will never be part of the optimal since $value[\{a_2,a_3\}]=36 < value[\{a_2\}]+value[\{a_3\}]=52$.

C	$value[C]$	C	$value[C]$	C	$value[C]$
$\{a_1\}$	33	$\{a_1,a_3\}$	87	$\{a_1,a_2,a_3\}$	97
$\{a_2\}$	39	$\{a_1,a_4\}$	70	$\{a_1,a_2,a_4\}$	111
$\{a_3\}$	13	$\{a_2,a_3\}$	36	$\{a_1,a_3,a_4\}$	100
$\{a_4\}$	40	$\{a_2,a_4\}$	52	$\{a_2,a_3,a_4\}$	132
$\{a_1,a_2\}$	87	$\{a_3,a_4\}$	67	$\{a_1,a_2,a_3,a_4\}$	151

Table 1: Coalition values for a CSG problem among 4 agents.

The DP[6] and IDP[7] are algorithms able to find an optimal solution to the CSG problem. They explore the complete solution space with complexity $O(3^n)$. IDP improves notably the performance and reduces the hardware requirements in comparison to DP. The structure of both algorithms is very similar, since IDP is a extended version of DP. In this section we describe first DP and then we present the improvements provided by IDP.

To do so, we will use the following terminology:

- Agent (a_x): A single agent, where x indicates the agent identifier.
- Agents (A): The set of all available agents. $A = \{a_1, a_2, \ldots, a_n\}$.

– Coalition (C): $C \subseteq A$. C is a subset of A that contains the agents participating in a coalition. Its size is defined as the number of agents forming the coalition.
– Split : The operation performing a binary partition of a coalition.
– Splitting : The result of the split operation. A splitting is a 2-tuple represented by (C_1, C_2), where $|C_1|$,$|C_2| > 0$, $C_1 \cap C_2 = \emptyset$.
– Coalition Structure (CS): A collection of disjoint coalitions whose union yields the entire set of agents. $CS \subseteq 2^A$ where for any $C_i, C_j \in CS, C_i$

2.1 The DP algorithm

Given the input data, DP first evaluates all the possible coalitions of size 2. After evaluating all the coalitions of a given size m, DP proceeds to evaluate all the coalitions of size $m + 1$. This process is repeated until m is equal to the size of the set of agents ($|A|$).

Algorithm 1 and Table 2 present the algorithm and a trace of a DP execution with the input data of Table 1.

The DP algorithm (Algorithm 1) activity can be characterized by three nested loops: (i) The outer loop, lines 1-14, where the coalition size is selected, (ii) the intermediate loop, lines 2-13, where the coalitions of a fixed size are generated, and (iii) the inner loop, lines 5-11, where every coalition is split and evaluated. Moreover, the first splitting in the inner loop is generated by the $getFirstSplit$ function in line 4 and the rest by the $getNextSplit$ function in line 10. Lines 7-9 asses the value of the best splitting, which is stored in memory in line 12.

Every time an m-sized coalition is selected, all the coalitions whose size is less than m have already been evaluated. Therefore their optimal values are known and DP can decide whether it is better to split the coalition or keep it as a whole. By using this strategy, DP ensures that at the end of its execution, the optimal coalition structure is found.

In table 1, the first column shows the size of the selected coalitions, which grows until reaching $|A|$. For each size m selected, DP enumerates all the possible coalitions C (second column) and for each coalition C it knows its associated coalition value $value[C]$ (third column). Next, DP enumerates all the possible splittings of each coalition (fourth column). Note that when $m = 2$, there is only one possible splitting, whereas for larger values of m, the number of splittings is also larger. Concretely, the total number of possible splittings for a coalition of size m is $2^{m-1} - 1$. For each splitting, the algorithm computes the sum of the splitting member's coalition values. If this is bigger than $value[C]$, the splitting value becomes the best coalition value $value[C]$. Last column shows the maximum value of a coalition and the coalition values of all its splittings.

2.2 The IDP algorithm

Although DP can find the optimal solution with a reasonable complexity, its memory requirements are large and as shown in [7] DP performs redundant calculation.

Algorithm 1 Pseudo-code of the DP algorithm

```
1: for m = 2 → n do
2:     for C ← coalitionsOfSize(m) do                    ▷ (n choose m) iterations
3:         max_value ← value[C]
4:         C1 ← getFirstSplit(C)
5:         while (C1) do                                 ▷ 2^(n−1) − 1 iterations
6:             C2 ← C − C1
7:             if (max_value < value[C1] + value[C2]) then
8:                 max_value ← value[C1] + value[C2]
9:             end if
10:            C1 ← getNextSplit(C1)
11:        end while
12:        value[C] ← max_value
13:    end for
14: end for
```

m	C	$value[C]$	Splittings (C_1, C_2)	Value of splittings $value[C_1] + value[C_2]$	max
2	$\{a_1,a_2\}$	87	$\{a_1\},\{a_2\}$	72	87
	$\{a_1,a_3\}$	87	$\{a_1\},\{a_3\}$	46	87
	$\{a_1,a_4\}$	70	$\{a_1\},\{a_4\}$	73	73
	$\{a_2,a_3\}$	36	$\{a_2\},\{a_3\}$	52	52
	$\{a_2,a_4\}$	52	$\{a_2\},\{a_4\}$	79	79
	$\{a_3,a_4\}$	67	$\{a_3\},\{a_4\}$	53	67
3	$\{a_1,a_2,a_3\}$	97	$\{a_1\},\{a_2,a_3\}$	85	
			$\{a_2\},\{a_1,a_3\}$	126	126
			$\{a_3\},\{a_1,a_2\}$	100	
	$\{a_1,a_2,a_4\}$	111	$\{a_1\},\{a_2,a_4\}$	112	
			$\{a_2\},\{a_1,a_4\}$	112	127
			$\{a_4\},\{a_1,a_2\}$	127	
	$\{a_1,a_3,a_4\}$	100	$\{a_1\},\{a_3,a_4\}$	100	
			$\{a_3\},\{a_1,a_4\}$	86	127
			$\{a_4\},\{a_1,a_3\}$	127	
	$\{a_2,a_3,a_4\}$	132	$\{a_2\},\{a_3,a_4\}$	106	
			$\{a_3\},\{a_2,a_4\}$	65	132
			$\{a_4\},\{a_2,a_3\}$	92	
4	$\{a_1,a_2,a_3,a_4\}$	151	$\{a_1\},\{a_2,a_3,a_4\}$	165	
			$\{a_2\},\{a_1,a_3,a_4\}$	166	
			$\{a_3\},\{a_1,a_2,a_4\}$	140	
			$\{a_4\},\{a_1,a_2,a_3\}$	166	166
			$\{a_1,a_2\},\{a_3,a_4\}$	154	
			$\{a_1,a_3\},\{a_2,a_4\}$	166	
			$\{a_1,a_4\},\{a_2,a_3\}$	125	

Table 2: Trace of execution of a problem of size 4.

IDP algorithm lessens this drawback.While DP generates all the possible splittings of each coalition, IDP [7] introduces conditions to avoid the generation

and evaluation of a large amount of splittings. IDP explores a fraction of the splittings explored by DP. This fraction only depends on the number of agents. For problems from 22 to 28 agents we found IDP to explore between 38% and 40% of the splittings explored by DP. Algorithm 2 presents the pseudo-code of IDP, which is very similar to the DP algorithm but altering lines 4-6 in order to filter what are the splittings to be evaluated. In this version, $IDPBounds$ function at line 4 assesses the bounds on the coalitions' size according to IDP description.

Algorithm 2 Pseudo-code of the IDP algorithm

```
1: for m = 2 → n do
2:     for C ← coalitionsOfSize(m) do                    ▷ (n choose m) iterations
3:         max_value ← value[C]
4:         (lower_bound, high_bound) ← IDPBounds(n, m)
5:         C1 ← getFirstSplit(C, lower_bound)
6:         while (sizeOf(C1) ≤ high_bound) do
7:             C2 ← C − C1
8:             if (max_value < value[C1] + value[C2]) then
9:                 max_value ← value[C1] + value[C2]
10:            end if
11:            C1 ← getNextSplit(C1, C)
12:        end while
13:        value[C] ← max_value
14:    end for
15: end for
```

3 Single-thread implementation

In this section we analyze the operations of generating and evaluating splittings inside the inner loop, which consume about 99% of the execution time. As seen in the algorithms 1 and 2 the inner loop is characterized by two actions: The splitting operation and a comparison using data fetched from memory. We analyze how these two actions are executed.

3.1 Splitting generation

The splitting generation problem can be reduced to the subset enumeration problem, since each coalition splitting is composed by a subset, C_1, and its complementary, C_2. Generating all the subsets C_1 from a coalition C and then calculating the complementary $C_2 = C - C_1$, though, would produce the same splitting twice: once for each of the splitting subsets. We remove one element from the coalition (the agent with the highest rank) when performing the subset enumeration, so that the removed element is never part of the enumerated subset and always belongs to its complementary.

There exist several ways of enumerating subsets [12], like banker's sequence, lexicographical order, and gray codes. Based on the description of IDP[7], the banker's sequence seems to be the enumeration technique employed by this algorithm, since it generates the splittings in growing order of $|C_1|$, and then simplifies the filtering of splittings by its size. Figure 1a shows a scheme of the banker's sequence operation for C=$\{a_1,a_4,a_5,a_6,a_7\}$, and assuming that only coalitions with $|C_1|$=2 need to be evaluated. Note that element a_7 is always assigned to the complementary subset (lighted colour). The generation starts directly from the first splitting of size $|C_1| = 2$, follows with the remaining $\binom{4}{2} - 1$ subsets of the same size, and stops before generating the first subset of size 3. The code does not waste instructions generating useless subsets.

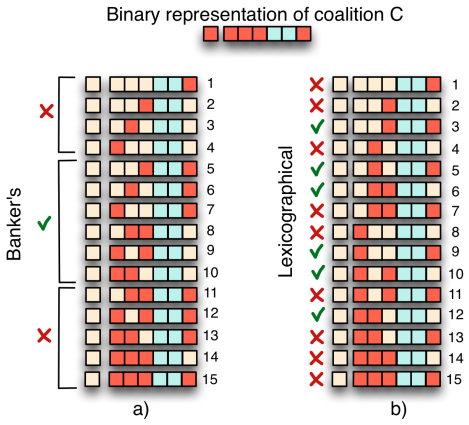


Fig. 1: a) Banker's sequence versus b) lexicographical order.

When generating splittings in lexicographical order (see Fig. 1b), some filtering code is required to check that the size of the splitting ranges between a given pair of bounds in order to fit IDP specification. Execution resources are wasted to generate splittings that are then discarded, and to perform the filter check. In Fig. 1, only 6 out of 14 splittings are actually needed (note the check and discard crossed signs).

Both methods were implemented using recurrent functions that calculate the next splitting from the previous one. The lexicographical order was implemented with a few number of very simple operations:

$$C_1 \leftarrow (C_1 + C^{**}) \text{ AND } C,$$

where C^{**} is the two's complement of C, that can be precalculated for all the splittings of a given coalition. The whole splitting code requires only 7 machine code instructions in a current x86 ISA. On the other hand, our implementation of banker's sequence, an improved version of the algorithm published in [12],

required, on average, 6 times more instructions. More details about the implementation, like the usage of a special population count instruction for computing $|C_1|$, can be found in the published code.

3.2 Memory accesses

The coalitions and their associated values are stored in a one-dimensional array, namely the values vector. A coalition is represented using an integer index where the bit at position x of the index indicates that agent x is a member of the coalition. The index determines the vector element containing the coalition value. Using this representation, the input of the CSG problem fits into a vector of 2^n-1 positions. With coalitions represented by 4-byte words, we can run problems up to 32 agents.

All memory accesses correspond to reads from the vector of coalition values performed in the inner loop of the algorithm, and a few writes on the intermediate loop. The total number of data read operations done by the DP algorithm is around 2×3^n. As explained above, IDP evaluates only a subset of the splittings, corresponding to 38%-40% of the read operations performed by DP.

The memory-level parallelism of the algorithm is moderate. The inner loop recurrence can generate multiple independent read requests, without having to wait for data, subject to storage availability for pending requests and for the window of instructions blocked on those data.

The data-reuse degree of the algorithm is high. There are 2^n elements in the *value* vector, and so the average number of reads to the same data item is $\approx 2\times(3/2)^n$ ($\approx 100,000$ for $n = 27$). However, accesses to the same item are scattered in time, specially when the algorithm analyzes medium- or large-size coalitions.

The bad performance behavior of the memory access pattern arises for vectors that do not fit into the processor's cache. The vector size is 2^{n+2} bytes, which is 16 MBytes for n=22. For larger n's an important amount of vector accesses will miss the cache and will request a full 64-Byte cache block to DRAM. This creates both latency and bandwidth problems. The moderate memory-level parallelism helps hiding part of the DRAM latency but, as we will show later, an important amount of this latency is exposed in the execution time.

4 Multi-thread implementation

This section analyzes the algorithm's data workflow in order to find its potential thread-level parallelism (TLP). Exploiting concurrency efficiently is not straightforward, and a new method to generate coalitions is devised. Finally, potential performance problems are described.

4.1 Identifying sources of parallelism

The simplest and most efficient approach is always to parallelize the outer loop of a program. DP and IDP, though, exhibit loop-carried dependencies on the

outer loop: the optimal values for coalitions of size m must be generated before using them for generating the optimal values for coalitions of size $m+1$.

The intermediate loop generates all the coalitions of a given size, and for each coalition it analyzes all the splittings of certain sizes. Tasks corresponding to coalitions are independent: they only modify the value associated to the coalition, and only read values corresponding to coalitions of lower size. Therefore, there cannot exist read-after-write (RAW) dependencies nor any other data dependence among the tasks. However, the single-thread code was designed to accelerate coalition generation by using an inherently sequential algorithm that uses the previous coalition to generate the next one in lexicographical order. The next subsection describes a method for breaking this artificial dependence.

4.2 Speeding up work distribution among threads

Assume we have t threads and we want each thread to evaluate a disjoint set of coalitions. We must distribute work to assure good load balance, and do it in a fast and efficient way. Table 3 illustrates the generation of all the possible coalitions of size m=3 from a set of n=6 agents. The single-thread code implements a sequential algorithm to generate in lexicographical order all $\binom{6}{3}$=20 coalitions, represented as bitmaps in the binary encoding columns of Table 3. In practice, we must calculate cnt=$\binom{n}{m}$ and then assign cnt/t coalitions to each thread. Once a thread obtains its starting position in the coalition series, say k, it can generate the whole range with the fast sequential method. But we need an efficient strategy to generate the k^{th} coalition without having to compute all the previous coalitions from the beginning.

Order (k)	Encoding Bin	Encoding Dec	Coalitions	Order (k)	Encoding Bin	Encoding Dec	Coalitions
1	...111	7	$\{a_1,a_2,a_3\}$	11	..111.	14	$\{a_2,a_3,a_4\}$
2	..1.11	11	$\{a_1,a_2,a_4\}$	12	.1.11.	22	$\{a_2,a_3,a_5\}$
3	.1..11	19	$\{a_1,a_2,a_5\}$	13	1..11.	38	$\{a_2,a_3,a_6\}$
4	1...11	35	$\{a_1,a_2,a_6\}$	14	.11.1.	26	$\{a_2,a_4,a_5\}$
5	..11.1	13	$\{a_1,a_3,a_4\}$	15	1.1.1.	42	$\{a_2,a_4,a_6\}$
6	.1.1.1	21	$\{a_1,a_3,a_5\}$	16	11..1.	50	$\{a_2,a_5,a_6\}$
7	1..1.1	37	$\{a_1,a_3,a_6\}$	17	.111..	28	$\{a_3,a_4,a_5\}$
8	.11..1	25	$\{a_1,a_4,a_5\}$	18	1.11..	44	$\{a_3,a_4,a_6\}$
9	1.1..1	41	$\{a_1,a_4,a_6\}$	19	11.1..	52	$\{a_3,a_5,a_6\}$
10	11...1	49	$\{a_1,a_5,a_6\}$	20	111...	56	$\{a_4,a_5,a_6\}$

Table 3: Coalitions generated using lexicographical order.

Algorithm 3 describes $getCoalition(n,m,k)$, a function that generates the k^{th} coalition in lexicographical order of m elements from a set of n. The description is done recursively to help understand how it works, although the actual implementation is iterative in order to improve its performance. The coalition is created recursively, bit by bit, starting from the least significant bit and considering $\binom{n}{m}$ possibilities. Around the first half of the possible coalitions have

the less significant bit set to 1. Concretely, if the requested rank, k, is lower than or equal to $h=\binom{n-1}{m-1}$, then the bit is set to 1, and m is decremented by one. Otherwise, the bit is set to zero, and the rank k is reduced to $k-h$. Each recursive call decrements the number of bits to consider to $(n-1)$.

Algorithm 3 pseudocode of $getCoalition(n, m, k)$

```
1: if ((m == 0) OR (k == 0)) then
2:     return 0
3: end if
4: h ← (n-1 choose m-1)
5: if (k ≤ h) then
6:     return 1 + 2×getCoalition(n − 1, m − 1, k)
7: end if
8: return 2×getCoalition(n − 1, m, k − h)
```

4.3 Potential parallel performance hazards

The first and last iterations of the outer loop exhibit few TLP, since the number of possible coalitions to be executed in parallel is determined by $\binom{n}{m}$, where n is constant and m is the iteration number. So either small and high values of m are compromising the efficiency of the parallel execution. We tuned the implementation so that threads are launched in parallel only for iterations that have a minimum amount of work. A minor problem is the need for a few number of synchronization barriers at the end of every iteration of the outer loop. They can be neglected, except for very small problem sizes.

An important performance issue is the occurrence of false cache sharing misses. They occur when different threads update different positions in the vector of values that happen to be mapped to the same cache line.

Finally, there is also the issue of true cache sharing. Threads generate values for coalitions of size m that are stored into local caches. When all the threads need to access those values for handling larger coalitions, data has to be moved from local storage to all the execution cores.

5 Experimental results

The computer system used in our experiments is a dual-socket Intel Xeon E5645, each socket containing 6 Westmere cores at 2.4 GHz, and each core executing up to 2 Hardware threads using hyperthreading (it can simultaneously execute up to 24 threads by Hardware). The Level 3 Cache or Last Level Cache (LLC) provides 12 MiB of shared storage for all the cores in the same socket. 96 GiB of 1333-MHz DDR3 RAM is shared by the 2 sockets, providing a total bandwidth

of 2×32 GB/sec. The Quickpath interconnection (QPI) between the two sockets provides a peak bandwidth of 11.72 GB/sec per link direction.

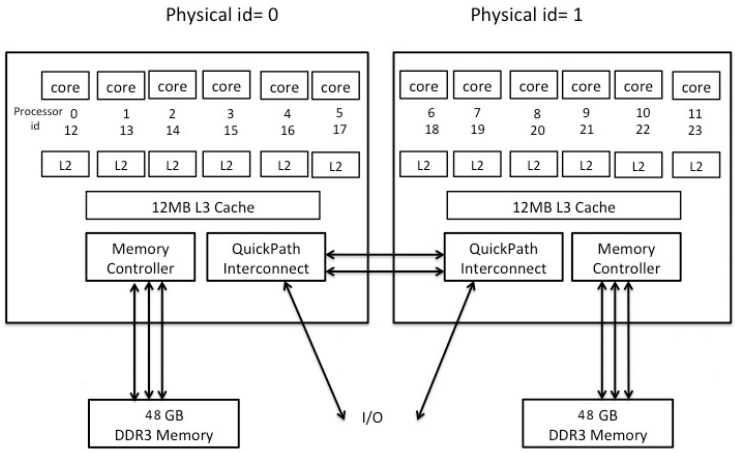


Fig. 2: Hardware configuration

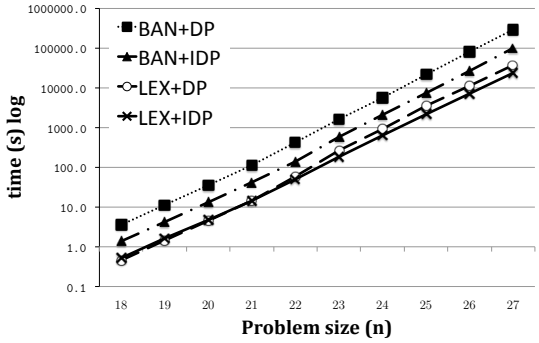
Figure 2 shows a diagram of the Hardware configuration used. Note, that this two-socket system has a Non-Uniform Memory Access (NUMA) configuration where half of the global memory is directly attached to one socket, and the other half to the other. Although for the programmer there is only one global memory system, some memory requests have to travel from one socket to the other, having its corresponding penalization in terms of CPU cycles.

Input data was created using a uniform distribution as described by [13] for problem sizes $n = 18 \dots 27$. In all the experiments, data is stored in the first memory system socket. This allows fast data access for all the threads in the single-core execution and for half of the threads in the multi-core execution.

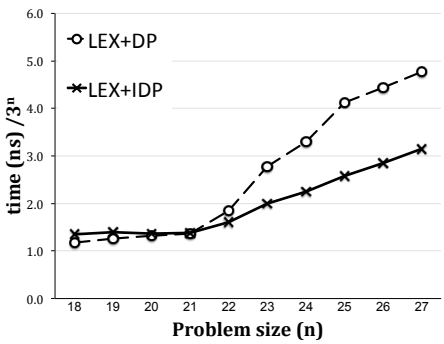
5.1 Single-thread execution

DP and IDP were executed using both the banker's and lexicographical splitting generation methods. Figure 3a plots the execution time in logarithmic scale for the four algorithmic variants. Splittings are computed around $7x$ to $11x$ faster using lexicographical order rather than banker's sequence. This is due to the fact that although the generation of the splittings using a banker's sequence generates less splittings, the lexicographical order generation is considerably faster. In fact, the number of instructions executed by the processor is around 6 times lower when using the lexicographical order technique.

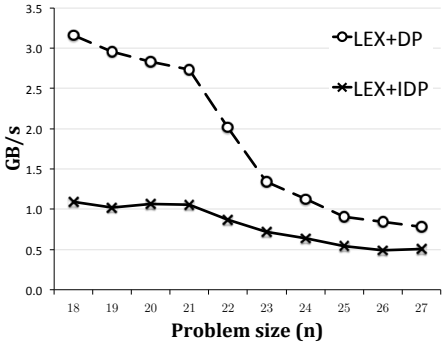
Figure 3b represents the execution time of DP and IDP divided by 3^n (algorithmic complexity). This metric evaluates the average time taken by the program to execute a basic algorithmic operation, in this case a splitting evaluation.



a) Execution time (log).



b) Time / Complexity $\Theta(3^n)$.



c) Effective Memory Bandwidth (GB/s).

Fig. 3: Experimental data (BAN: Banker's sequence; LEX: Lexicographical order).

It is similar to the CPI (Cycles Per Instruction) metric, but at a higher level. The metric helps identifying performance problems at the architecture level. Figure 3b shows two different problem size regions: those that fit into the last level cache (n<22), and those that do not. A small problem size determines a computation-bound scenario, where DP slightly outperforms IDP, even when it executes around 20% more instructions. The reason is that IDP is penalized by a moderate number of branch mispredictions.

Large problem sizes determine a memory-bound scenario, where IDP amortizes its effort on saving expensive memory accesses to outperform DP by 40-50%. Figure 3c shows the effective memory bandwidth consumption seen by the programs. The shape of the curves can be deduced from Figure 3b, but we are interested on the actual values. The effective bandwidth ranges between 0.5 and 1.0 GB/sec. A small fraction of this bandwidth comes from the last level and lower-level caches, and the remaining fraction comes from DRAM. Even considering the worst case described in section 3.3, that only 4 bytes out of the 64-Byte cache block are effectively used, it is still a very small value compared

to the peak 32 GB/sec. The conclusion is that DRAM latency is the primary performance limiter. Results on the next subsection corroborate this conclusion.

5.2 Multi-thread execution

We focus our multi-thread analysis on IDP, which outperforms DP for interesting problem sizes. We run IDP using t= 6, 12, and 24 threads. The case t=6 corresponds with using a single processor socket. The case t=12 uses only one socket but also exploits its hyperthreading capability. Finally, t= 24 is an scenario where all 2 sockets have their 6 cores running 2 threads each, using hyperthreading. Figure 4 shows the speedup compared to the single-thread execution. Again, distinguishing between small and big problem sizes is useful.

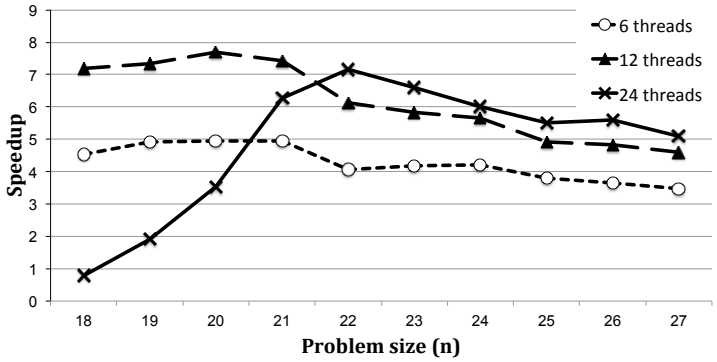


Fig. 4: Single-thread IDP versus 6-, 12- and 24-thread IDP execution

The t=6 configuration provides a speedup of 5 for small problems, and lower than 4 for large problems. The t=12 configuration further increases performance around 60% for small problems, and 30% for bigger problems. The fact that executing two threads per core does improve performance corroborates previous latency limitations, since hyperthreading is a latency-hiding mechanism. It also indicates that 6 threads do not generate enough cache memory and DRAM requests to fully exploit the available cache and DRAM bandwidth.

The effective memory bandwidth achieved with 12 threads is around 2.5 GB/sec for the bigger problem sizes, or around 13 times lower than the peak achievable bandwidth. Given the lack of spatial locality of DRAM accesses, we are probably reaching the maximum bandwidth available for the pseudo-random memory access pattern of the problem.

The t=24 configuration checks the benefit of using a second socket. Performance is highly penalized for small problems, due to the overhead of communication traffic along the QPI links for both false and true cache sharing coherence. On average, half of the data accessed by a thread is fetched from the other

socket. Compared to the single-socket scenario, where all data is provided from local caches, performance drops up to 7 times for very small problems.

Large problems benefit very little from a second socket, with improvements of nearly 10%. The advantage of the 2-socket configuration is that the available DRAM bandwidth is duplicated, and the overhead due to coherence traffic is not so important, given that most of the data is obtained from DRAM. Although a benefit, the small performance gain does not justify using a second socket. Again, the symmetric, scattered memory access pattern does not fit well with the NUMA hierarchy. We are currently working on a way to partition data that reduces communication between sockets.

6 Conclusions and future work

This paper presents an optimized implementation of the DP and IDP algorithm and a novel contribution describing the first parallel version of DP and IDP.

Our implementations clearly outperform the state-of-the-art DP and IDP algorithms. According to [14], they need 2.5 days to solve a CSG problem with 27 agents[4]. We obtain same order results (same problem in 27 hours) using a single-threaded execution and a banker's sequence as the splitting generation algorithm.

Our best single-threaded implementation -using lexicographical order- solves a same sized CSG problem in 5.8 hours and the multi-thread implementation reduces execution time to 1.2 hours. Our implementation provide a significant improvement over reported results and we have made available to the community our source code.

We have analyzed the bottlenecks of DP and IDP. The pseudo-random memory access pattern lacks locality, and exploits the memory system capabilities very inefficiently. The latency tolerance ability of multi-threading improves performance on a multi-core processor. However, although a dual-socket NUMA system is offering the better result for big problems, investing the same amount in a single-socket multicore-system could provide a higher speedup than using a a dual-socket NUMA architecture. The use of GPUs or accelerators with massive thread parallelism will be analyzed in the future.

We also want to study alternatives for coalition indexing and storage that provide higher locality, even at the expense of increasing instruction count, which is not a performance limiter for large problems.

7 Acknowledgements

This research has been supported by MICINN-Spain under contracts TIN2011-28689-C02-01, TIN2012-38876-C02-01 and the Generalitat of Catalunya (2009-SGR-1434).

[4] in some unspecified computer and using a code implementation that is not provided.

References

1. Shehory, O., Kraus, S.: Methods for task allocation via agent coalition formation. Artif. Intell. **101**(1-2) (1998) 165–200
2. Sandholm, T.W., Lesser, V.R.: Coalitions among computationally bounded agents. Artificial Intelligence **94** (1997) 99–137
3. Rassenti, S., Smith, V., Bulfin, R.: A combinatorial auction mechanism for airport time slot allocation. The Bell Journal of Economics (1982) 402–417
4. Voice, T., Ramchurn, S.D., Jennings, N.R.: On coalition formation with sparse synergies. In: AAMAS. (2012) 223–230
5. Sandholm, T., Larson, K., Andersson, M., Shehory, O., Tohmé, F.: Anytime coalition structure generation with worst case guarantees. Arxiv preprint cs/9810005 (1998)
6. Yun Yeh, D.: A dynamic programming approach to the complete set partitioning problem. BIT Numerical Mathematics **26** (1986) 467–474 10.1007/BF01935053.
7. Rahwan, T., Jennings, N.R.: An improved dynamic programming algorithm for coalition structure generation. In: AAMAS (3). (2008) 1417–1420
8. Rahwan, T., Ramchurn, S., Jennings, N., Giovannucci, A.: An anytime algorithm for optimal coalition structure generation. Journal of Artificial Intelligence Research **34**(1) (2009) 521–567
9. Rahwan, T., Jennings, N.: Coalition structure generation: dynamic programming meets anytime optimisation. In: Proceedings of the 23rd Conference on Artificial Intelligence (AAAI). (2008) 156–161
10. Michalak, T., Sroka, J., Rahwan, T., Wooldridge, M., McBurney, P., Jennings, N.: A distributed algorithm for anytime coalition structure generation. In: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1, International Foundation for Autonomous Agents and Multiagent Systems (2010) 1007–1014
11. Sutter, H.: The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobb's Journal **30**(3) (2005) 202–210
12. Loughry, J., van Hemert, J., Schoofs, L.: Efficiently enumerating the subsets of a set. http://www.applied-math.org/subset.pdf (2000)
13. Larson, K.S., Sandholm, T.W.: Anytime coalition structure generation: an average case study. J. of Experimental & Theoretical Artificial Intelligence **12**(1) (2000) 23–42
14. Rahwan, T., Ramchurn, S.D., Dang, V.D., Giovannucci, A., Jennings, N.R.: Anytime optimal coalition structure generation. In: AAAI. (2007) 1184–1190